1987

A study of the programming language APL /

Larry S. Musolino
Lehigh University

A Study of the Programming Language APL

by

Larry S. Musolino

A Thesis

Presented to the Graduate Committee

of Lehigh University

in Candidacy for the Degree of

Master of Science

in

Computer Science

Lehigh University

1987

This thesis is accepted and approved in partial fullfilment of the requirements for the degree of Master of Science.

May 6, 1987
date

Samuel L. Gulick
Professor in Charge

Donald J. Hillman
Head of Divison

L. J. Vanucci
Chairman of Department

Table of Contents

Abstract

A Programming Language (APL) was originally developed as a convenient notation to concisely express many of the important algorithms in mathematics. It later was transformed into an implementation for various IBM computers, and is now available on a wide range of computer systems. In many respects, the APL language is unparalleled in its conciseness, elegance and power. For various reasons, however, the language has never achieved the widespread usage and interest once envisoned.

Advocates of APL maintain that the language allows the user to conceptualize the problem at hand, and not have to concern oneself with the tedium of performing iterations to manipulate similar data elements. In addition, it is argued that the APL user need know very little about the underlying computer architecture.

APL critics consider the language to be cryptic, terse, and slow due its interactive nature.

This study of APL will attempt to briefly describe the language, and investigate the reasons for its limited usage.

1.0 APL Beginnings

The programming language APL was developed in the late 1950's as a notation for expressing ideas about computation. Ken Iverson, the original developer of APL, originally defined the notation at Harvard University and later refined it at IBM Corporation. The acronym APL was derived from the title of the book "A Programming Language" originally written by Iverson in 1962. This text essentially contained the original definition of the language.

Iverson's original intent was not to develop a conventional programming language as much as to develop a convenient notation to concisely express many of the important algorithms in mathematics. As a result, the original APL definition utilized a variety of special symbols, conventions and notations such as subscripts, superscripts, and a two-dimensional program syntax with arrows designating flow of control that made the notation extremely difficult to implement on a computer. However, the original version of APL was a useful and powerful conceptual tool for the precise and concise statement of algorithms. In many cases, a complex programming problem was first written in APL and then translated by hand into a more conventional programming language. As an example, the original APL language was used to provide a concise and complete formal description of the IBM 360 computer hardware.

The decision to implement APL as a computer language was motivated by the desire of Iverson and Adin Falkoff, manager of the APL design group at IBM, to use APL as a teaching and system design tool. A significant stumbling block, however, was (and still is) the uniqueness of the APL character set and the inability of popular CRT terminals to display many of the important characters.

In the early 1960's, an IBM subsidiary was planning to market the language, but was unsuccessful because the subsidiary choose an acronym unacceptable to Falkoff and Iverson. Between 1966 and 1968, the language was available mostly within IBM, but pressure from outsiders who had seen the language and wanted to use it led the company to support it officially. In the mid 60's an experimental APL time-sharing system for the IBM System 360 became available within IBM and is now an IBM program product. Between 1968 and 1970 several companies were formed to offer APL timesharing services, such as Scientific Timesharing Corp, Rockville, MD. An organization of APL users is the Association of Computing Machinery (ACM) special-interest group on APL (SIGAPL). This group sponsors seminars and conferences regarding APL use and applications. The New York chapter of SIGAPL annually conducts a seminar entitled "APL: A Tool of Thought".

APL has attracted a devoted group of users and evoked much controversy in its lifetime. Iverson's original notation is remarkable in its conciseness, power and elegance. This power and conciseness makes the

language especially well-suited to the interactive environment, because a line of only a few characters can accomplish a surprising amount of computation. In many cases the body of an APL subprogram consists of a single line. In fact, an anonymous APL programmer once commented on the power of APL: "It took God seven days to create the world, but an APL programmer could do it in one line".

In APL the basic or *primitive* operations generally accept whole arrays as arguments and produce whole arrays as results. Thus the basic unit of data in APL tends to be an entire array rather than a single array element as in languages such as FORTRAN and Pascal. This emphasis on array processing gives programming in APL a unique style, which is quite different from any other language.

APL has many advantages over other programming languages. The APL user needs to know very little about the underlying computer architecture and internal representations of program and data. The user can concentrate on expressing data manipulations precisely and avoid the tedium of writing iteration to perform the same operations on similar data elements. The tools provided by APL are often much closer to the way the programmer conceptualizes the program. Because of this simplicity and conciseness, it is not uncommon for an APL program to require only one-tenth as many statements as a more conventional language.

The philosophy of APL is best summarized by Iverson, who states [Ref. 4] : "the systematic treatment of complex algorithms requires a suitable programming language, and such a programming language should be concise, precise, consistent over a wide area of application, mnemonic, and economic of symbols; it should permit the description of a process to be independent of the particular representation chosen for the data." Iverson goes on to discuss the notational aspect of APL : "most of the concepts and operations needed in a programming language have already been defined and developed in one or another branch of mathematics. Therefore, much use can and will be made of existing notations. However, since most notations are specialized to a narrow field of discourse, a consistent unification must be provided. For example, separate and conflicting notations have been developed for the treatment of sets, logical variables, vectors, matrices and trees, all of which may ... occur in a single algorithm."

2.0 APL Program Format and Simple Examples

The first step to programming in APL is to become familiar with the special symbols that make up the language. The APL character set is made up of the capital letters of the alphabet and certain other arithmetic symbols and Greek letters. Figure 1 contains a representation of the APL keyboard. The special APL symbols normally not found on "standard" keyboards include:

¨	≥	≤	≠	÷	×	⍒
_	α	∈	ρ	ω	ι	φ
↓	↑	→	←	○	•	⍱
⊂	⊃	∪	∩	∇	∆	⊥
⍞	⌈	⌊	⊢	⊣	~	∨
⍁	⍋	⍑	⍟	∧	⍲	

In addition, other symbols can be formed using the overstrike feature of APL. The actual keyboard uses some logic in the placement of special symbols on the APL keyboard, such as ω over W , ϵ over E , ρ (rho) over R , * (for power) over P , ○ (circle symbol) over O , α over A , ⌈ (for ceiling) over S , ' (for kwoute) over K, etc. Although lowercase letters are not allowed in APL, fifty-two letters are available through the use of the underline character (shift "F").

APL operates in two modes. The first is immediate execution mode,

where each well-formed APL expression is evaluated after the line is typed. The second is function or programming mode where a group of APL expressions is defined under a function name. When this name is typed, the entire group of expressions is then executed sequentially. When a user first logs onto an APL system, they are placed in immediate execution mode, and a *workspace* is cleared.

The special symbols found on the APL keyboard are called primitives. They are the basic building blocks of APL functions. Many are similar to arithmetic operators, such as $+$, $-$, $\times$, $\div$. Other primitives are denoted by ρ , ι , ϵ , etc. Most APL primitives are dyadic, that is, they require both a right and left argument. APL also utilizes monadic functions which operate on a single argument, always to the right of the primitive.

Before proceeding with an introduction to the various APL capabilities, there are certain idiosyncrasies associated with APL which should be noted:

- the symbol corresponding to arithmetic subtraction, $-$, is distinct from the symbol denoting a negative number. The latter is referred to as the "high minus" , $^{-}$, and is used to show that a number is negative.

- **expressions are evaluated from right to left,** with no dependence on operator precedence, thus $20 \times 2 - 3 + 4$ evaluates to $^{-}100$, not

41. Parentheses may be used in the normal manner to indicate order of evaluation.

2.1 Simple Primitives

APL has a large number of primitive functions which are already defined for the user. In addition, user-defined functions can easily be added to the APL workspace. The built-in primitive functions are similar to the functions available on a hand-held calculator. For example, typing 5 + 4 produces the result 9 as:

```
      5 + 4
9
```

Notice that the APL prompt is an indentation of five spaces. APL allows the both integers and real numbers to be intermixed in any operation:

```
      1.0 + 3.0
4

      3.5 × 2.6
9.1
```

As an example of the distinction between the arithmetic subtraction primitive, −, and the symbol to indicate a negative number, ¯, consider the following:

```
      37 − ¯7
44
```

```
      ¯3 - ¯2
-1
      ¯6
-6
-6
```

The table below lists some built-in APL functions and the equivalent conventional form for the expression.

APL Form	Conventional Form	Description
A + B	a + b	Addition
A − B	a − b	Subtraction
A × B	a × b, or ab	Multiplication
A ÷ B	a ÷ b, $\frac{a}{b}$, or a/b	Division
A * B	a^b	a raised to the b power
A ⍟ B	$\log_a$ b	Base a logarithm of b
B * .5	$\sqrt{b}$ or $b^{½}$	Square root of b
⍟ B	$\log_e$b or ln b	Natural logarithm of b
÷ B	1 ÷ b or $b^{(-1)}$	Reciprocal of b
|B	|B|	Absolute value of b
!B	b!	Factorial of b
* B	e^b or exp(b)	e to the b power

In general, APL primitives are either monadic (one argument) or dyadic (two arguments). A somewhat confusing aspect associated with several primitives is that the special symbol which defines the function can have

two completely different meanings dependent on whether it is used monadically or dyadically. For example, |B gives the absolute value of B while A|B gives the remainder of dividing B by A, so that:

```
      |3 ¯1 2
3 1 2

      1|3.4
0.4

      3|6 7 8 9
0 1 2 0
```

Of course it is also possible to define a niladic function (no arguments) such as:

```
      ∇Z ← TOSS
[1] Z ← +/?6 6 ∇

      TOSS
9

      TOSS
5
```

This function simulates the tossing of two dice by choosing two random numbers between 1 and 6 and summing them. More concerning function definition is presented in Section 2.17, but briefly the arrow (←) denotes assignment, ∇ begins a function and a second ∇ closes the function definition. By typing the name of the function, the statements in the definition are then executed sequentially.

2.2 Assignment of Variables

The previous expressions can be immediately evaluated at an APL terminal provided that variables have been assigned values. The assignment operator is logically designated by the left arrow symbol, ←, so that A ← 4 assigns the value 4 to A. Of course, the right side of the assignment can be an expression so that B ← 3 × 2 × 5 assigns 30 to the variable B.

Variable names can be made up of any combination of letters (A to Z), underscored letters (A to Z), digits (0 to 9), or the symbols Δ or Δ, but **cannot** begin with a digit or with SΔ or TΔ, nor contain any spaces. Since spaces are not allowed, the delta character is frequently used to separate parts of a variable name, such as LASTΔVALUE. Variable names can be as long as 66 characters on most APL systems.

Variables can also be assigned to non-numeric data, such as character strings, by enclosing the data in single quotes.

```
EMPLOYEE ← 'XY SMITH'
SALARY ← '1000'
PHONE ← '123-4567'
```

Character data and numeric data may not be intermixed in APL operations, so that if a character variable was defined to be the character representation of a number

```
      A ← '600'
```

and this was to be added to a numeric variable

```
      B ← 400
      A + B
```

the system would respond with a DOMAIN ERROR message to indicate an invalid operation.

When a variable is assigned a value, nothing is printed. The value can be called for display by typing its name.

```
      PI ← 3.14159
      PI
3.14159
```

If a variable name without an assigned value is used in an expression, APL will complain and attempt to pinpoint the approximate position of the error.

```
      2.71828 × ABC
VALUE ERROR                    (* Error Message *)
2.71828 × ABC                  (* Expression is printed
          ^                        with error pointer *)
```

An APL function never alters its argument so that typing A + 1 does not increase A, but simply prints the value of A + 1. Assignment is needed to change the value of A, as in A ← A + 1.

A user comment in APL is declared through the use of the so-called "thumbnail" symbol (⍝) as follows:

```
⍝ This is the data from the experiment
A ← 3.1 4.5 6.7 8.2 6.99 3 0.22
```

2.3 Vectors

The power and flexibility of APL is apparent when one considers the available manipulations of vectors and matrices. No special considerations are needed to define a vector or matrix. For example, to define a six element vector and assign it to a variable VEC6ELEM:

```
VEC6ELEM ← 1 4 9 16 25 36
```

The numbers separated by blanks constitutes a vector. It is possible to add vectors, element by element, as in:

```
2 3 5 + 6 2 9
8 5 14
```

Vectors must be of the same length for the addition to be valid, or APL will complain:

```
1 2 3 + 4 5 6 7
LENGTH ERROR                    (* Error Message *)
1 2 3 + 4 5 6 7
^                               (* Error pointer *)
```

A basic concept in APL is that of scalar extension. When certain

operations are attempted on a vector or matrix with a scalar, APL automatically extends the scalar to the same dimension as the array. For example

```
      3 × 10 20 30
```

really equates to

```
      3 3 3 × 10 20 30
```

As mentioned previously, APL distinguishes the operator to indicate arithmetic subtraction from the operator to indicate a negative scalar (high minus). The distinction becomes obvious in the following example:

```
      VECTOR1 ← 3 5 - 1 2
      VECTOR1
2 3

      VECTOR2 ← 3 5 ¯1 2
      VECTOR2
3 5 -1 2
```

In the first example, APL interperts the expression as the vector 1 2 subtracted from the vector 3 5 producing the result 2 3.

2.4 Maximum and Minimum ⌈ ⌊

APL provides maximum and minimum operations for both monadic and dyadic use. If used monadically, maximum and minimum are referred to as ceiling and floor. Ceiling rounds any decimal number up to the nearest integer. Floor rounds any decimal number down to the nearest integer. As examples of the monadic use:

```
      ⌈144.3
145
      ⌊9.9999
9
```

These operations also apply to vectors of any length:

```
      ⌈5.8 12.01 15 ¯675.4
6 13 15 ¯675
      ⌊8.0 ¯7.6 12.9 5
8 ¯8 12 5
```

When maximum and minimum are used dyadically, the result is the max or min of two arguments. If the arguments are vectors the corresponding elements are compared, and of course, the number of elements in the two vectors must be equal.

```
      100⌈101
101
      75⌊¯1
¯1
      17.5 ¯6 12.0 6⌊ 17 79 17 3
17 ¯6 12 3
```

2.5 Rho ρ

The rho operation is sometimes referred to as the *shape* operator when

used monadically since it returns the number of elements in a vector.

```
      VEC6ELEM ← 1 4 9 16 25 36
      ρVEC6ELEM
6
```

When ρ is applied to character variables, the number of characters between the single quotes (including spaces) is returned.

```
      DATE ← 'DECEMBER 12, 1985'
      ρDATE
17
```

When used monadically ρ returns the shape of a vector. When used dyadically ρ can be used to shape a vector into a matrix.

2.6 Iota ι

When ι is used monadically it produces a vector of consecutive integers from one up to and including the right argument.

```
      ι8
1 2 3 4 5 6 7 8
```

The starting value for ι can be changed to be zero instead of one. This is accomplished by a system command which alters a system variable called *index origin*. System commands are discussed in Section 2.16.

The argument to ι must always be a non-negative integer. This does not mean that sequences of negative integers cannot be created since ι can be

combined with other arithmetic operators.

```
      ¯1 × ⍳5
−1 −2 −3 −4 −5

      12 + 5 × ⍳4
17 22 27 32
```

Virtually any sequence of numbers can be created by combining ⍳ with other available APL functions. The functions ⍳ and ⍴ can be combined to return the position of elements in a vector.

```
      VEC5ELEM ← 100 300 500 700 900
      ⍳⍴VEC5ELEM
1 2 3 4 5
```

2.7 Matrices

APL is especially well-suited for creating and manipulating matrices. All functions which apply to vectors can be used in dealing with matrices. A simple way to create a matrix is to use the ⍴ operator dyadically. The left argument to ⍴ is the number of rows and columns in the matrix, separated by a space. The right argument is the data to be shaped.

```
      MATRIX2BY4 ← 2 4⍴10 20 30 40 50 60 70 80
      MATRIX2BY4
10 20 30 40
50 60 70 80

      MATRIX4BY2 ← 4 2⍴10 20 30 40 50 60 70 80
```

```
      MATRIX4BY2
10 20
30 40
50 60
70 80
```

When used monadically ρ will return the shape of the matrix.

```
      ρMATRIX2BY4
2 4
      ρMATRIX4BY2
4 2
```

When a matrix is created with the ρ operator, the elements to the right of ρ are read in order for each row from left to right until the matrix is filled. If there were too few numbers to the right of ρ, the numbers would be repeated to fill the matrix. If there were too many numbers, the matrix would be filled and the remainder would be discarded.

```
      NOTENUF ← 2 3ρ200 400 600 800
      NOTENUF
200 400 600
800 200 400

      NOTNOTENUF ← 2 2ρ1 2 3 4 5 6 7 8
      NOTNOTENUF
1 2
3 4
```

Any of the primitives which work on vectors can also be applied to matrices in the same way.

```
      3 + 2 × NOTENUF
 403 803 1203
1603 403  803
```

In the same way that scalar extension functions, the scalars in the previous example are extended to correspond to the matrix to be operated on. If matrices are defined to be the same size then any of the scalar dyadic functions can be applied just as they are to vectors.

In dealing with matrices it is important to be able to access individual elements. In order to do this APL use brackets to indicate the index of an array, similar to the Pascal method, so that

```
      VEC4ELEM ← 10 20 30 40
      VEC4ELEM[3]
30

      VEC4ELEM[1 3]
10 30
```

The index may be repeated to obtain the element more than once.

```
      VEC4ELEM[3 4 1 1 1 2 2 4 3 3]
30 40 10 10 10 20 20 40 30 30
```

In order to index into a matrix, it is necessary to specify both the row and column positions. This is accomplished by using a semicolon to separate row and column. All index numbers to the right of the

semicolon indicate row numbers, while all numbers to the left of the semicolon indicate column numbers.

```
      NEWYORK ← 3 4⍴'SMOG TRAFFIC'
      NEWYORK
SMOG
 TRA
FFIC
      NEWYORK[1;4]
G
```

To return a portion of the second row:

```
      NEWYORK[2;2 3 4]
TRA
```

To return an entire row, the row number is entered and a blank after the semicolon:

```
      NEWYORK[3; ]
FFIC
```

To return several columns, a blank is left in the row position and the desired column numbers entered:

```
      NEWYORK[ ;2 3]
MO
TR
FI
```

An interesting fact of indexing is that the result which is returned is

always one dimension higher than the indexing argument, so that if a scalar is used as an index, the result is a vector and if a vector is used as an index, the result is a matrix.

2.8 Relational Operators $< \leq = \neq \geq >$

These operators are used in the standard manner to compare two numbers or several numbers in an array. APL returns the values of true and false as the integers 1 and 0 respectively.

```
      17 > 98
0
      ¯12.1 ≤ 12.1
1
```

The relational primitives can be used on vectors and matrices in the usual way:

```
      87 ¯7 13 = 13 13 13
0 0 1
      12 13 14 15 16 = 11 + ⍳5
1 1 1 1 1
      MAT2BY4 ← 2 4⍴2 × ⍳3
      MAT2BY4
2 4 6 2
4 6 2 4
```

```
      MAT2BY4 = 6

0 0 1 0

0 1 0 0

      4 ≤ MAT2BY4

0 1 1 0

1 1 0 1
```

2.9 Logical Operators Λ V ~

The three logical operators, AND (Λ), OR (**V**), NOT (~), can operate only on the Boolean arguments 0 and 1. The truth tables are the standard Boolean tables. The main application of these logical primitives is in conjunction with the relational primitives, such as:

```
      (5 > 6)  V (991 = 991)

1

      ~ 5 ≥ 3

0
```

These logical operators also have important applications in reduction and compression, discussed in the following sections.

Listed below is a table of several of the scalar dyadic functions, together with their meanings:

Expr	Meaning
A $\lceil$ B	Maximum of A and B
A $\lfloor$ B	Minimum of A and B
A | B	A- Residue of Y (remainder)
A $<$ B	A Less Than B
A $\leq$ B	A Less Than or Equal to Y
A $=$ B	A Equal to B
A $>$ B	A Greater Than B
A $\geq$ B	A Greater Than or Equal to B
A $\neq$ B	A Not Equal to B
A $\wedge$ B	A And B
A $\vee$ B	A Or B

2.10 Compression /

Compression is a feature unique to APL and is used to select from an array of numbers. Compression is similar to the AND function where a 1 is used to select a value and a 0 is used to compress or de-select a number.

```
      1/65
65

      0/7

(empty line)
```

Compression is most often used on arrays or vectors. The length of the compression vector of 0's and 1's must be equal to the length of the

argument or APL will complain with a LENGTH ERROR message. Scalar extension can also be used with the compression primitive.

```
      1 1 0 0 1/12 87 ¯5 18.8 23.1
12 87 23.1
```

Note that the length of the returned vector is less than the length of the argument since the 0's in the second and third position of the left vector have "compressed" the corresponding values in the right argument.

```
      0/1 2 3 4 5

(empty line)

      1 1 0 0/0 0 0 0
0 0
0 0

      1 1 0 0/1 2 3 4 5
LENGTH ERROR
1 1 0 0/1 2 3 4 5
^
```

Some versions of APL extend this feature where the left argument is not restricted to a Boolean value, but can be any integer. The left argument then indicates the number of times the corresponding right argument is to be "duplicated".

```
      4/5 6 7
5 5 5 5 6 6 6 6 7 7 7 7

      4 3 2 1 0/10 20 30 40 50
10 10 10 10 20 20 20 30 30 40
```

2.11 Reduction

One of the most important primitives in APL is reduction, which combines compression with other APL primitives, such as +, ×, −, ÷, etc. Reduction is denoted by an APL primitive placed to the left of the compression slash, as in +/. The effect of reduction is to apply the primitive, such as +, between each pair of the members of the argument vector. For example, plus reduction can be used to sum a vector.

```
      +/12 14 16 20
62
```

This produces the same result as 12 + 14 + 16 + 20.

```
      FACTORS ← 2 3 5 7 11 13
      ×/FACTORS
30030
```

Reduction may also be applied to matrices. Plus reduction of a matrix would result in the sum across each individual row.

```
      RED3BY2 ← 3 2⍴⍳6
      RED3BY2
1 2
3 4
5 6

      +/RED3BY2
3 7 11
```

To get the sum of the individual columns an axis operator can be specified after the slash. This would indicate which dimension specified in the shape the reduction is to operate on.

```
      +/[1]RED3BY2
9 12
```

If a 2 was entered in the brackets, the results would be the sum across the individual rows, since column is the second parameter of shape, and summing the values of each column is equivalent to the sum of each row.

To sum up all the element in a matrix, the plus reduction can be taken of the plus reduction. The first reduction would be the sum of the individual rows; the sum of the rows would then be the sum of the matrix.

```
      +/+/RED3BY2
21
```

The maximum and minimum primitives can be combined with compression to form maximum reduction (⌈/) and minimum reduction (⌊/). These operators work the same as other reduction operators in that the result is the same as if the primitive was inserted between each pair in the vector or array.

```
      ⌈/12 3 ¯4 87
87

      ⌊/10 20 ¯5 ¯90 54
¯90
```

It should be noted that the comparison is done from right to left and the result of each successive pair is compared with the next pair.

Reduction can also be combined with the logical operators. However, since the arguments to the logical primitives can be only Booleans, the right argument to the logical reduction operator must consist of Booleans, or expressions which evaluate to Booleans.

```
      ∧/1 1 1 1 1 1
1

      ∧/1 1 1 1 0 1
0

      ∨/0 0 0 0 0 0 0 1
1
```

The result is equivalent to the insertion of the logical primitive between each element of the right argument.

2.12 Take and Drop ↑ ↓

The APL primitives, Take and Drop, return modified versions of their right argument. The take primitive returns as many elements from the right argument as is specified in the left argument. The drop primitive drops as many elements in the right argument as is specified in the left argument. If the left argument is a positive integer, it indicates the operation is to start at the beginning of the vector, while a negative

integer indicates the starting point is the end of the vector.

```
      CHAR∆DATA ← 'THIS IS A STRING'
      NUM∆DATA ← 11 13 17 19 23 29 31
      4↑CHAR∆DATA
THIS
      ¯8↑CHAR∆DATA
A STRING
      ¯9↓CHAR∆DATA
THIS IS
      3↑NUM∆DATA
11 13 17
      ¯4↑NUM∆DATA
19 23 29 31
      6↓NUM∆DATA
31
```

Using take and drop on matrices is a bit more involved since the left argument has to specify both rows and columns. Thus the left argument requires a two-element vector.

```
      MATRIX3BY4 ← 3 4⍴⍳12
      MATRIX3BY4
1 2 3 4
5 6 7 8
9 10 11 12
```

```
      3 2 ↑ MATRIX3BY4
1 2
5 6
9 10

      1 4 ↑ MATRIX3BY4
1 2 3 4

      ¯2 3 ↑ MATRIX3BY4
5 6 7
9 10 11
```

When using ↓ with matrices, a 0 may be used to mean "none".

```
      2 0 ↓ MATRIX3BY4
9 10 11 12

      0 3 ↓ MATRIX3BY4
4
8
12
```

If the left argument in take is greater than the number of elements in the right argument, the positions will be zero-filled for numeric data or blank-filled for character data.

```
      TOOSMALL ← 2 2⍴⍳4
      TOOSMALL
1 2
3 4
```

```
      33↑TOOSMALL
120
340
000
```

2.13 CONCATENATE ,

The comma is used to concatenate or join data objects of the same type. The data objects may be scalars, vectors or matrices.

```
      123,456
123456

      ABC←123456
      ABC,789
123456789

      DEF←54321
      ABC,DEF
123456543 21
```

2.14 Grade Up and Grade Down ⍒ ⍋

These functions are formed by using the overstrike feature of APL. When used with a vector right argument, grade up returns a list of the positions of the elements in the vector in the order necessary to put the vector in ascending order.

```
      Δ 100 36 76 2 19 54
4 5 2 6 3 1
```

Grade up does not return the sorted vector, but instead returns the order in which to sort the vector. In the above example, the fourth element would go first, the fifth element second, the second element third, etc. In order to actually produce the sorted array, the grade up could be used to index on the original vector:

```
      UNSORT ← 100 36 76 2 19 54
      UNSORT[Δ UNSORT]
2 19 36 54 76 100
```

The grade down is similar except that it returns a list of the positions of the elements in the vector in the order necessary to put the vector elements in descending order.

```
      ∇ UNSORT
1 3 6 2 5 4
```

To put the vector in descending order, the same approach can be used:

```
      UNSORT[∇ UNSORT]
100 76 54 36 19 2
```

Notice that, in general, the list resulting from grade up is the reverse of that resulting from grade down

2.15 Other APL Primitives

The preceding APL primitives are a fraction of the total number available. Several others are presented here, with several examples.

The APL primitive epsilon, , is used to determine if the left argument to ϵ is a member of the right argument.

```
      100 ϵ 10 20 30 40 70 90 100
1

      'H' ϵ 'ABCDEFG'
0
```

The APL primitive Ø is used to transpose a matrix along a diagonal.

```
      MATRIX ← 3 5ρι15
      MATRIX
1 2 3 4 5
6 7 8 9 10
11 12 13 14 15

      Ø MATRIX
1 6 11
2 7 12
3 8 13
4 9 14
5 10 15
```

The APL primitive ⌽ is used to rotate a matrix along the last axis specified by the left argument. The matrix itself is given as the right argument to ⌽.

```
      1 ⌽ MATRIX
2 3 4 5 1
7 8 9 10 6
12 13 14 15 11
```

The APL primitive ? is used to generate a random number from a iota right argument. If used dyadically, ? produces *n* non-repeating random numbers based on the left argument *n*.

```
      ?20
6
      ?20
14
      ?20
9
      3 ? 20
2 19 11
```

The monadic function pi-times (○) gives a result of pi (π) times the argument used where π is taken to be approximately 3.141592654. For example:

```
      ○ 1 2 .5
3.141592654  6.283185307  1.570796327
```

If the circle function is used dyadically, as in R ← A ○ B, the argument A determines which circular (trigonometric) function is to be applied to

the argument B. All angles are expressed in radians. (The monadic operator ◯ can be used to convert degrees to radians, as in ◯ THETA ÷ 180). As is customary, the arc sine, arc cosine and arc tangent functions return as results the principal values, since many different angles would be equally correct. The table below defines the various circular functions which are available:

A	A ◯ B	Restrictions
1	sin B	
2	cos B	
3	tan B	
4	(1 + B * 2) * .5	
5	sinh B	
6	cosh B	
7	tanh B	
0	(1 − B * 2) * .5	1 ≥ |B
¯1	arcsin B	1 ≥ |B
¯2	arccos B	1 ≥ |B
¯3	arctan B	1 ≥ |B
¯4	(¯1 + B * 2) * .5	1 ≤ |B
¯5	arcsinh B	
¯6	arccosh B	B ≥ 1
¯7	arctanh B	1 > |B

2.16 System Commands

The APL environment in which data are stored and calculations performed is called a *workspace*. When a user first logs on, the APL workspace is automatically cleared. The APL system is then available for

interactive use or function definition (discussed in next section). In order to save work from one APL session and use it in another session, APL allows a user to "save"the current workspace for later recall. In order to save a workspace the system command **)WSID NAME** can be used. (APL system commands begin with a right parenthesis). The command WSID indicates the current workspace should be named "NAME', where NAME is any valid APL variable name. The save command can then be used to save the named workspace, **)SAVE NAME** . Most systems will respond with the time and date, identifying that the workspace was indeed saved. The complimentary system command to recall an old workspace into the current workspace is **)LOAD NAME**.

Once a workspace has been loaded , any of the previously defined functions can be modified, deleted, etc. Once the newly modified workspace is ready to be saved a simple **)SAVE** will write the current workspace over the previous one.

If several workspaces have been created a **)LIB** command will produce a list of workspaces.

An entire workspace can be eliminated using the command **)DROP NAME** . Functions or variables can be copied from one workspace to another by using the **)COPY** command in the form **)COPY WSNAME FUNCTION1 .. FUNCTIONN VAR1 .. VARN,** where WSNAME is the workspace name to be copied from and

FUNCTION1..FUNCTIONN and VAR1..VARN are the list of functions and variables to be copied.

To get a listing of the current variables, the command **)VARS** will print all the variable names defined in the current workspace. To remove variable names, and their values, the command **)ERASE VAR1 VAR2 ...** will erase variables VAR1, VAR2, etc. To erase the contents of the entire APL workspace type

```
)CLEAR
CLEAR WS                (* APL response *)
```

A list if all functions that have been defined in the current workspace can be obtained by typing the system command **)FNS.**

A number of system variables are also defined as part of the current workspace. In general, the value of these variables can be modified with system commands. Several system variables are given below, together with their meanings and default values:

Variable Name	Significance	Default Value
ORIGIN	Starting point for counting and indexing	1
WIDTH	Line width used for output	80
DIGITS	Maximum number of significant digits shown in output	10
SYMBOL	Maximum number of allowed variable names in current workspace	256
FUZZ	Comparison tolerance used in equality testing	1 E ¯13

2.17 Function Definition ∇

An important aspect of APL programming is function definition, analogous to procedure definition in Pascal. The term function is borrowed from mathematics and connotes the concise, algorithmic quality of the APL language.

In order to define functions, the ∇ symbol is used, followed by the name of the function.

∇ ETOTHEX

The APL system will recognize the beginning of a function definition and prompt for each line of the function definition. A final ∇ closes the definition.

[1] 'ENTER VALUE FOR X'

```
[2] X ← □
[3] ANSWER ← * X
[4] 'FOR X =', ⍕ X
[5] 'THE VALUE OF EXP(X) IS' , ⍕ ANSWER
∇
```

As in line [1], if a string is enclosed in quotes, it will simply be printed when the function is executed. When a □ is encountered, line [2], the system will wait for user input. The input can be a scalar, vector, matrix, etc. Whatever is entered will be assigned to the variable X. Line [3] assigns to the variable ANSWER the value of exp(X). Line [4] contains the symbol ⍕ which is called the *thorn*. It is used to temporarily convert the numeric variable X to a character variable so it can be concatenated (,) to the characters which precede it. If ⍕ were not used APL would report a DOMAIN ERROR when it tried to concatenate a character and numeric variable.

To run the function, the function name is typed:

```
ETOTHEX
ENTER VALUE FOR X  1.0
FOR X = 1.0
THE VALUE OF EXP(X) IS 2.7182818
```

To see the entire function ETOTHEX a combination of ∇ and □ is used :

```
∇ ETOTHEX [□] ∇
[1] 'ENTER VALUE FOR X'
```

```
[2] X ← □
[3] ANSWER ← * X
[4] 'FOR X =', ⍕ X
[5] 'THE VALUE OF EXP(X) IS' , ⍕ ANSWER
∇
```

If it is desired to edit the function, a variety of options are available. In most case the function must first be "opened" by typing ∇ and the function name. The system can be directed to a particular line by typing the line number in brackets.

```
[4]
```

A particular line may also be requested and a specific character position in the line specified. A slash can then be used to delete the current position and a comma used to replace the deleted character.

```
∇ ETOTHEX [4□12]
[4] 'FOR X =', ⍕ X
      /,EQUALS
[4] 'FOR X EQUALS', ⍕ X
```

Function editing can be quite different on various APL systems. Some systems provide a sophisticated editor, while others provide a bare minimum.

It is desirable to have the capability to define variables within a function to be local, rather than global. This would cause variables which have been defined local to a certain function to have values only when the

function is being executed. At other times, the variables are unbound. To define local variables it is necessary to list them in the header of the function:

```
∇RES ← X0 AVERAGE X1
[1] SUM ← X0 + X1
[2] RES ← SUM ÷ 2
∇
```

In this example, X0 , X1 and RES are local variables. Notice that since the function AVERAGE is defined to be dyadic, it must be provided with the proper number of arguments:

```
      7 AVERAGE 13
10
      100 400 AVERAGE 2 2
51 201
      AVERAGE 10 12 13
SYNTAX ERROR
AVERAGE 10 12 13
          ^
```

In the previous example, a newly defined temporary variable is generally considered to be global. To have this be a local variable instead, it is necessary to include it in the function header separated by semicolons.

The function header can contain an arbitrary number of arguments. The form of the function header takes on a different appearance depending

on the number of arguments. The table below gives examples of function headers for 0, 1 and 2 arguments:

	Number of Arguments		
	0	1	2
Explicit Result	Z←TOSS	Z←DIVS B	Z←A MIDPT B
No Explicit Result	PER	TEST B	A PRINT B

2.18 Branching

Ordinarily, the statements of a function are performed sequentially. APL provides an interesting method of branching within functions, using the branch arrow (→), followed by APL expression. If the result of the expression is an integer, then the statement having that integer as its line number is executed next. If the result of the expression is null, no branch is taken, and the line following the branch statement is executed next. A domain error results if the expression gives a noninteger value. A vector of integers can can also be used and the first element of the vector is used to determine which line will be executed next. As shown below, several "tricks" are possible:

[3] → 5

(Line 5 would be executed next)

[3] → 2 + 2 × A = B

(If A=B, line [4] would be executed. If A ≠ B line 2 would be executed)

[3] → 0/5

(Line [4] would be executed next, since 0/5 produces an "empty" vector)

[3] → (K < 4)/2

(If K<4 then 1/2 selects line number [2] else if K≥2 then 0/2 compresses 2 and the next line ([4]) would be executed)

If the APL expression to the right of the branch arrow evaluates to an integer value which is not a valid line number in the function, then the function is exited. This is sometimes used as a method for error handling, etc.

3.0 Reasons for Limited Use of APL

When one considers the question of APL in relation to other computer languages, a fundamental distinction must be recognized : APL is **not** a computer language. It is a general purpose mathematical notation, developed independently of computers, which has been slightly adapted so that its expressions can be evaluated by computers. This distinction points out the invalidity of direct comparisons with conventional programming languages.

APL is generally not considered a mainstream computer language as is C, Pascal, FORTRAN, Cobol, etc. In fact, it is unlikely that the majority of students and programmers have more than heard of APL. Several reasons for this lack of widespread usage are discussed.

An obvious factor in the limited usage of APL is its special character set. For the most part the standard CRT terminal is unable to handle the special character set, and in the early days, Iverson and Falkoff were insistent on retaining it.

An additional factor is that APL code can be cryptic. As discussed, mathematical symbols are used to replace most of the functions and commands found in more conventional languages. Because every symbol stands for an operation, a single page of APL could theoretically encode a program that would take thousands of lines in a more conventional

language, such as C.

For example, the expression

$$(2 = + / \varnothing\ 0 = (\iota \lfloor N * .5) \circ . | \iota N) / \iota N$$

is an APL program for generating all primes less than N. APL advocates, on the other hand, insist that this cryptic quality affords a conciseness that lets APL users think about complex algorithms without getting lost in do-loops, array subscripts, and other programming details.

As cited previously, APL was not developed to be a computer language as such. Instead, it was developed as a notation for expressing ideas about computation and algorithms.

Another reason for APL's relative obscurity is its interactive nature. It is difficult to find much basis for this; perhaps programmers view an interperted language to be akin to a "non-sophisticated" language, e.g. BASIC. Critics cite a slowness of execution related to the interactive nature of APL. What is not mentioned however, is that a single line of APL code can accomplish far more than in other languages. Furthermore, certain APL implementations actually produce code which executes faster than would its compiled equivalent in a language such as C.

This aspect, though, may tie in to a philosophical barrier that APL, or

for that matter, any unique language, must overcome to gain acceptance. Programmers who learn a specific language and have done a considerable amount of software design and coding in that language may lose a great deal of common sense when approached with a new or different language, no matter how advanced, powerful, etc. the new language may be. This is especially true of early FORTRAN programmers.

In summary then, several reasons for the limited usage of APL are as follows:

- Specialized Keyboard
- Cryptic Nature of APL Code
- Inability to use Standard CRT terminals
- Interactive Nature of APL Code
- Non-Traditional Aspect of APL Language

4.0 APL Applications

On the surface, it may appear that APL has very little to offer to the commercial user. The notation can be very terse, the language is interperted rather than compiled, and specially adapted keyboards and printers are required. In general, however, APL's use as a system design tool for planning, decision support and management information is nearly

unlimited. This is supported by the wide range of various APL applications.

The gamut of APL applications range from banking, brokerage, image processing, database operations to medical applications, such as radioactive decay statistics and biology population models. Indeed, it is an indication of the diversity and enthusiasm of APL adherents that one-of-a-kind scientific applications are continually cropping up.

In addition, APL is still regarded as the language of choice in a small but important area in the computer field: the information center. This is a department, typically within a medium or large corporation, that writes application software to perform tasks needed within the company. The tasks are generally specific to the company so the software is not available commercially. APL's compact code and logical, interactive nature are a big plus in the information center, because they allow experienced programmers to write applications quickly. In addition the matrix emphasis of APL makes the language well suited for manipulating large amounts of tabular data, a common task in applications written in information centers.

An interesting APL application is that of Citibank NA, headquartered in New York. The organization's Management Profitability Reporting System, written entirely in APL, tracks activities at more than 1000 of the bank's offices in over 100 countries and provides monthly profit

analyses for the bank's senior management. The success of the reporting system has led the bank to create a similar system for its credit card division. This program helps bank officials decide whom to invite to apply for credit cards. The APL program predicts the overall profitability that might be expected with an account, using such parameters as monthly balances, account buildup, interest rates and maintenance costs. In a benchmark comparison at Citibank, a forecast that took **two days** to execute using the Lotus Symphony spreadsheet program on an IBM personal computer took only **three minutes** with the APL code on the same computer.

The versatility and flexibility of APL is apparent when one examines the nature of recent APL applications [Ref. 1]:

- manufacturing planning systems (MRP)
- corporate planning models
- library automation
- nuclear measurements
- quality business management
- statistics
- graphics representation and analysis of space-based measurements

- decision support systems
- system design and prototyping
- logic programming
- artificial intelligence
- typesetting

5.0 APL at Lehigh University

There are no special terminals available at Lehigh University which possess the APL character set. Several DEC LA36 terminals were previously available , however these have been eliminated due to their incompatibility with the campus-wide network.

APL can be used on a standard terminal through the substitution of mnemonics and ASCII characters for the various APL functions. An example would be the use of the mnemonic .RO for the APL function ρ. Additionally, an "escape" character can be used with various uppercase characters. Thus, to obtain the ρ function, either .RO can be used or @R where @ is regarded as an escape character. No delimiting blanks are necessary and the two modes can be intermixed freely.

To use APL on the DEC-20, a user types 'aplsf' in response to the system level prompt, which is @. APLSF then responds with a terminal designator in the form: terminal...

If the user is unsure of what to respond, a H (for Help) should be entered, and a list of valid terminal designators is produced. The terminal type which is selected is important, not for input since the keyboard or escape mode can be freely intermixed, but for output where a mode must be defined and adhered to; the **)MODE** system command allows you to select the output mode.

A listing of the various APL functions, together with the DEC-20 keyboard mode format and escape mode format is presented below:

APL Set	TTY Set	Description	Escape Mode
∧	&	logical and	
←	_	assignment	
*	*	exponentiate	@P
×	#	multiply	
÷	%	divide	
'	'	quote string	@K
?	?	question (randomize)	@Q
↑	^	take	@Y
|	.AB	residue (absolute value)	@M
α	.AL	alpha	@A
□	.BX	quad (box)	@L
⌈	.CE	ceiling (maximum)	@S
↓	.DA	drop	@U
⊥	.DE	decode	@B
∇	.DL	del	@G
∩	.DU	down under	@C
⊤	.EN	encode	@N
∈	.EP	epsilon	@E
⌊	.FL	floor	@D
→	.GO	goto (branch)	
ι	.IO	iota	@I
Δ	.LD	delta	@H
○	.LO	circle	@O
⊃	.LU	left union	@X
≠	.NE	not equal to	
−	.NG	negation	
ω	.OM	omega	@W
ρ	.RO	rho	@R
⊂	.RU	right union	@Z
∪	.UU	up union	@V

It is somewhat disconcerting to use the mnemonics in dealing with the

APL System. After using APL for some time, the programmer views the special symbols of the APL character set as a natural method of expression. However, this is a mechanism which allows the use of the APL system in the absence of a specialized APL terminal, and in general, computer scientists must sometimes make do.

References

1. APL86 Conference Proceedings, APL Quote Quad, Volume 16 No. 4, Manchester England, July, 1986.

2. Gilman, L and Rose, A. APL - An Interactive Approach, 2nd ed., John Wiley and Sons, 1976.

3. Katzan, H APL Programming and Computer Techniques, Van Nostrand Reinhold, 1970.

4. Iverson, K. A Programming Language, John Wiley and Sons, 1962.

5. Pakin, S. APL : A Short Course, Prentice Hall, Inc 1973.

6. Polivka, R. and Pakin, S. APL - The Language and its Usage, Prentice-Hall, 1975

7. Smillie, K. APL360 with Statistical Examples, Addison-Wesley, 1974.

8. Smith, A. APL - A Design handbook for Commercial Systems, John Wiley and Sons, 1982.

9. Wiedmann, C. Handbook of APL Programming. 1st ed. Mason and

Lipscomb, Inc. 1974.

Vita

Larry S. Musolino is an engineer in the Quality Assurance organization at AT&T Technology Systems, Allentown, PA. He has written several technical papers regarding early life reliability of electronic components. He has received a B.S.E.E. degree cum laude from the City College of New York (CCNY) in 1980 and a M.S.E.E. degree from Lehigh University in 1983. He is a member of Tau Beta Pi, Eta Kappa Nu and the National Society of Professional Engineers (NSPE).

A Study of the Programming Language APL

Larry S. Musolino

Abstract

A Programming Language (APL) was originally developed as a convenient notation to concisely express many of the important algorithms in mathematics. It later was transformed into an implementation for various IBM computers, and is now available on a wide range of computer systems. In many respects, the APL language is unparalleled in its conciseness, elegance and power. For various reasons, however, the language has never achieved the widespread usage and interest once envisoned.

Advocates of APL maintain that the language allows the user to conceptualize the problem at hand, and not have to concern oneself with the tedium of performing iterations to manipulate similar data elements. In addition, it is argued that the APL user need know very little about the underlying computer architecture.

APL critics consider the language to be cryptic, terse, and slow due its interactive nature.

This study of APL will attempt to briefly describe the language, and investigate the reasons for its limited usage.